ARMY RESEARCH LABORATORY

Interfacing a Microelectromechanical System (MEMS) Sensor Array for Traumatic Brain Injury Detection with a Microcontroller

by Timothy C. Lee and Luke J. Currano

ARL-TR-6240 **October 2012**

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

Army Research Laboratory
Adelphi, MD 20783-1197

ARL-TR-6240 **October 2012**

Interfacing a Microelectromechanical System (MEMS) Sensor Array for Traumatic Brain Injury Detection with a Microcontroller

Timothy C. Lee and Luke J. Currano
Sensors and Electron Devices Directorate, ARL

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED (From - To)
October 2012	Final	

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Interfacing a Microelectromechanical System (MEMS) Sensor Array for Traumatic Brain Injury Detection with a Microcontroller	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)	5d. PROJECT NUMBER
Timothy C. Lee and Luke J. Currano	5e. TASK NUMBER
	5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
U.S. Army Research Laboratory ATTN: RDRL-SER-L 2800 Powder Mill Road Adelphi, MD 20783-1197	ARL-TR-6240

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Traumatic brain injuries (TBIs) result from exposure to high accelerations and are a serious threat to Soldiers in close contact with improvised explosive devices as well as sports players who are frequently involved in collisions. To improve TBI detection, the U.S. Army Research Laboratory (ARL) has developed a sensor small enough to be mounted in the ear. The sensor consists of an array of 3-axis microelectromechanical system (MEMS) acceleration threshold switches with different sensitivities that move to contacts under acceleration and complete a circuit. Previously, the outputs, which were voltage levels, required an analog-to-digital converter, but the implementation of the mechanism introduced a delay of 100 µs between samples. This delay has caused the loss of data from switch closures that last less than 100 µs, so the sensor was redesigned with digital outputs, and a new program was developed. Clocked signals were used to simulate sensor data, and tests showed improved delays of 3 µs to 10 µs.

15. SUBJECT TERMS

Traumatic, brain, injury, microcontroller

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Luke J. Currano
Unclassified	Unclassified	Unclassified	UU	38	19b. TELEPHONE NUMBER (*Include area code*) (301) 394-0566

Standard Form 298 (Rev. 8/98)
Prescribed by ANSI Std. Z39.18

Contents

List of Figures

Acknowledgments

I wish to acknowledge Dr. Luke Currano and the members of the Microelectromechanical System (MEMS) and Energetics laboratory, Marvin Conn and Arthur Harrison.

Student Bio

I am a third-year undergraduate student attending the University of Maryland at College Park. I intend to graduate with a double degree in electrical engineering and math. I hope to enter the rising energy sector and develop efficient technologies to improve energy consumption.

1. Introduction/Background

Traumatic brain injuries (TBIs) are a common result of high acceleration events and pose a health threat to Soldiers exposed to improvised explosive devices as well as sports players who are frequently involved in collisions. While severe TBIs may cause obvious superficial damage, mild TBIs that might occur several times are harder to detect and diagnose. There are existing sensors in the military and civilian sports that are intended to detect and classify the severity of impacts using commercially available accelerometers or pressure sensors (or both) (*1*, *2*). The main limitation of current acceleration and pressure sensors is that they require a constant source of power during monitoring.

To minimize power consumption, researchers at the U.S. Army Research Laboratory have developed an array of 3-axis microelectromechanical system (MEMS) acceleration threshold switches to detect acceleration events (*3*, *4*). On a single 3 mm x 3 mm chip, there are five spiral-shaped springs, each of which are compliant in the *x*-, *y*-, and *z*-axes that close an electrical circuit when an acceleration event exceeds a designed threshold, as shown in figure 1.

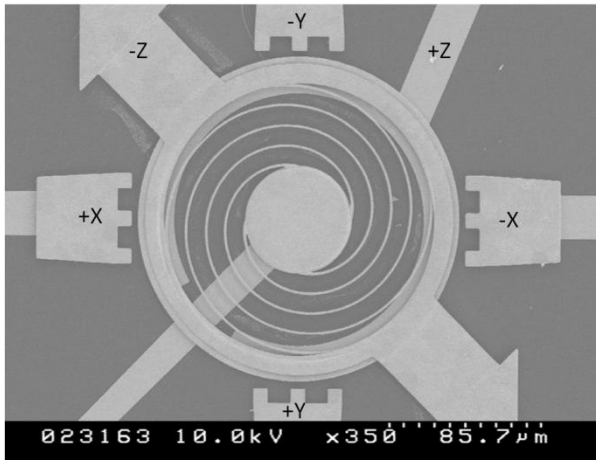

Figure 1. Scanning electron micrograph (SEM) of a Generation 3 sensor; a spiral spring sensor completes a circuit when an acceleration threshold is exceeded.

While the device is idle, the circuits are open, so no power is required except during actual events. When a switch is closed, current travels through a resistor network, and the output voltage is read across a reference resistor. The lowest output voltage level is read when the lowest threshold switch is closed, and the highest output voltage level is read when the highest threshold switch is closed. This setup reduces the number of outputs from 31 (+X, –X, +Y, –Y, +Z, and –Z for each of five levels plus ground [GND]) to 7 (six voltage readings and GND). The tradeoff is that the processor must analyze voltage levels, which are analog values. The analog-to-digital converter (ADC) takes a minimum of approximately 20 µs for each conversion (*5*).

The previous program implemented the ADC multiple times per sampling cycle (once for each output), for a total of a 100-µs delay between samples.

Device testing included short bursts of accelerations from a shock tube to simulate the shock wave from an explosive blast. The device's response included many individual switch closures, some of which lasted fewer than 100 µs. The delay was causing significant data loss, so the device was redesigned with digital outputs and an increased number of threshold levels. The outputs of this new version would not be routed through a resistor network. Instead, the raw value of 0 or 3.3 V would be sent to the microcontroller, which would read these values as a "0" or "1." When an acceleration threshold was exceeded, the mass would hit three contacts, and send a POS or NEG, an X, Y, or Z, and a level 1–8, as shown in figure 2.

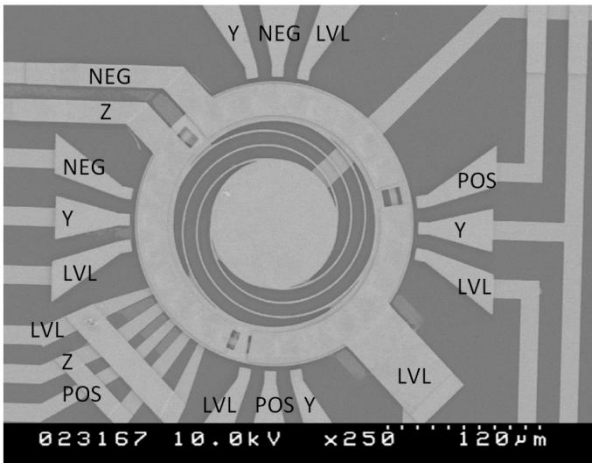

Figure 2. SEM of a Generation 4 sensor. The spiral spring sensor touches three contacts when an acceleration threshold is exceeded. Duplicate signals are tied together to reduce outputs.

As such, the number of outputs was increased to 14 (X, Y, Z, POS, NEG, Level 1, 2, 3, 4, 5, 6, 7, 8, and GND). The new digital design eliminates the need for the ADC, and the time and power overhead associated with it. The previous Texas Instruments (TI) microcontroller was also replaced with a newer and faster model. This report discusses the development of the new program to interface the sensor array with the new TI microcontroller.

2. Hardware and Initial Testing

The main hardware that was being tested was the TI CC430F5137 microcontroller. The chip was in a 48-pin package soldered onto an EM430F5137RF900 target board, which included a Joint Test Action Group (JTAG) connector, a radio frequency (RF) antenna socket, and fully accessible pins to provide external signals to the ports, as shown in figure 3. The necessary

drivers were installed to the computer through provided TI software, and the microcontroller was connected via a MSP-FET430UIF USB Flash Emulation Tool. Programs were written in Code Composer Studio, version 5.

Figure 3. EM430F5137RF900 target board.

A series of test programs were written to test the functionality and capability of the microcontroller. Some programs were used from TI's provided Code examples and from an introductory textbook on microcontrollers (*6*). The initial programs were simple: flashing the onboard light-emitting diodes (LEDs) continuously at a single frequency with a software loop, using multiple timers to flash the LEDs at different frequencies, and checking for a button input via a software poll.

Once some basic functionality was shown, interrupts were incorporated. Interrupts, which are a major component in control algorithms, trigger on specific actions like a button press or a timer overflow and redirect the program flow to appropriate subroutines. Software loops that continuously poll for input changes and use software delays are power intensive because the central processing unit (CPU) is constantly running. Using interrupts, the CPU can be shut down during idle times, which reduces power consumption, a major goal of the final design.

To conclude the initial testing phase, a demo program was written that incorporated basic concepts like timer usage, input handling, and data processing as well as more advanced concepts like RF transmission, data storage in both RAM and flash, and data retrieval. The demo program was to retrieve a stream of binary numbers and send them to a receiver via RF. The receiver was to store the data into flash. Then, using switches, the receiver would parse through the data in both directions and display the binary number on LEDs. The pseudocode for the demo is shown in figure 4, and the code is shown in appendix A.

```
main(){
     initialize LEDs and radio operation
     loop forever {
          enter low_power_mode
          wake up on one of three conditions:
               1) Button is pushed on Port 1 (input)
               2) Radio receives a signal
               3) Button is pushed on Port 2 (output to
                  LED)
     }
}
1) Button is pushed on Port 1 {
     Transmit byte via RF
}
2) Radio receives a signal {
     Toggle blinking LED
     Write to flash
}
3) Button is pushed on Port 2 {
     Cursor through flash memory in desired direction
     Display binary number on LEDs
}
```

Figure 4. Pseudocode outlining the steps of the demo program.

3. Results and Final Program

The final program had to meet several design specifications. The CPU was to enter a low power mode while no input changes were read. Assuming the inputs were initially 0, a rising edge on any of the pins would trigger an interrupt, which would wake up the processor. When awake, the time recorded by the real-time clock would be stored. All following input changes and the time elapsed between samples would be recorded. A software poll, as opposed to interrupt method, was used to collect data once the processor was awake, to avoid latency issues because each interrupt service introduces a delay of about 6 µs. Data collection would be stopped 500 ms after the last input change, which is longer than the duration of a typical acceleration event. The samples would be stored into flash memory and transmitted via RF, after which the program would re-enter the low power mode. Pseudocode outlining the steps of the program is shown in figure 5. The code for the latest version (version 4) is shown in appendix B.

```
main(){
      initialize the inputs/leds, the real-time clock, the
      sample bank, the flash memory, and the clock
      loop forever {
            enter low_power_mode
            wake up on any input rising edge change
            store the real-time clock value
            process the data via software poll and store in
            RAM. Repeat until 500 ms of no data
            store the data into FLASH
            transmit via RF
            clear variables
      }
}
```

Figure 5. Pseudocode outlining the steps of the final program.

3.1 Data Structure

There were two data structures considered for storing data from the 13 digital inputs. In the first design, a binary label 0001-1101 (corresponding to the numbers 1 through 13 in decimal) would be assigned to each pin. When an input change was detected, the program would determine which of the 13 pins changed. A flag, indicating whether the pin was high or low after the change, was appended to the label, and the 5 bits of data were stored in a byte, as seen in figure 6. With this configuration, every change on a pin is recorded with a byte. However, this also means that during each sampling cycle, each of the pins have to be individually checked and handled. In the best-case scenario, no changes occur; no extra processing must occur, so the cycle time is 4 µs. In the worst case scenario, a change is recorded on all 13 pins, which then requires 13 bytes to be stored (1 byte per pin). The sampling cycle in this case increases to 25 µs, primarily because of the time required to write the data to RAM.

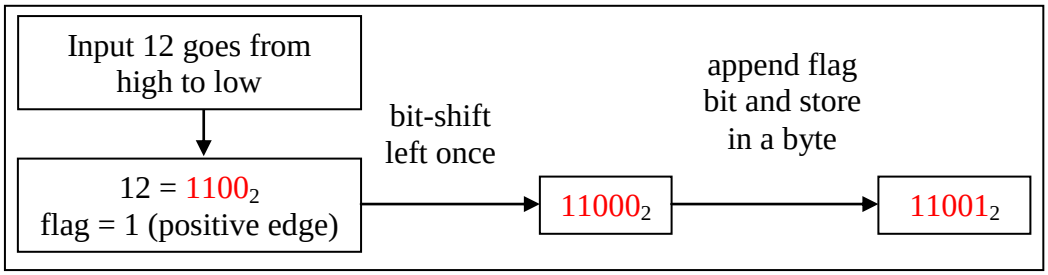

Figure 6. Flowchart indicating data processing for an input change in input 12. The final result is stored in a byte in RAM.

A second data structure removed the sampling cycle time's dependency on the number of input changes. The inputs read from the ports during each cycle were bit-shifted and masked so that each pin corresponded to a single bit. With 13 input pins, this meant 13 bits were required to store the data. The ports were set up with sensor inputs connected to pins 1–7 on port 1 and

pins 1–6 on port 2. When a change was detected, a logic statement determined which pins changed from high to low and vice versa, as shown in figure 7. The drawback of this model was that four bytes of data had to be stored whenever even a single input changed—as opposed to the first model, which stored only one byte per changed input. However, in this model no more than 4 bytes of data will ever need to be written from one polling cycle. In contrast, the first model could require writing as many as 13 bytes of data. With respect to timings, the model had a best-case scenario where no change was detected and each sampling cycle takes 4 µs. The worst-case scenario was independent of how many inputs changed. If any input changed, the cycle time would be 10 µs. Although this is not as fast as the first design in the case of one or two inputs changing, it is likely that in the final application of the sensor, multiple directions would be experiencing different accelerations in a very short period of time. Therefore, the second design has a smaller chance of data loss than the first.

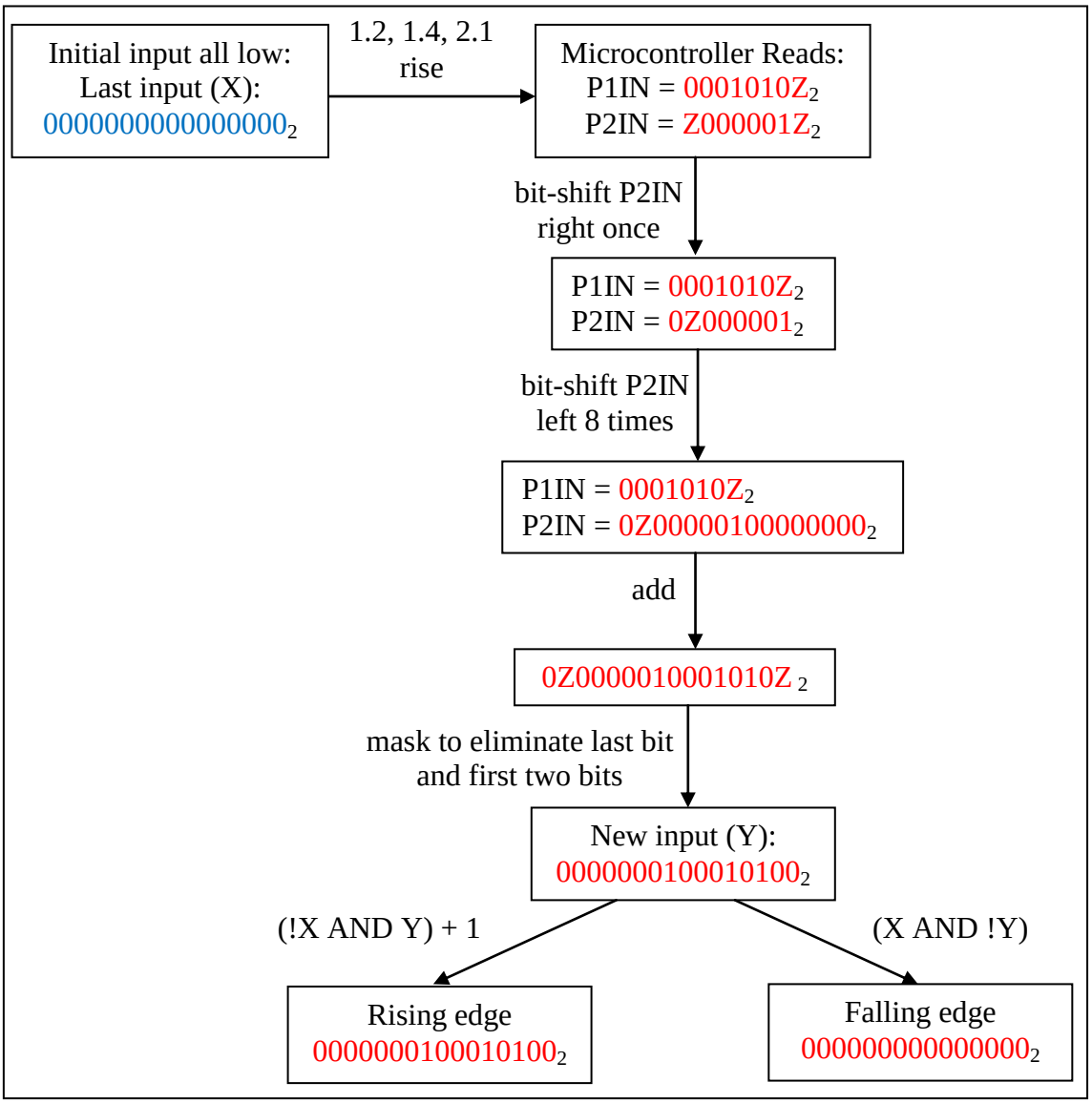

Figure 7. Flowchart depicting data processing in the second data structure. The notation A.B refers to port A, pin B. The Z's indicate signals that can be any value. By using Boolean logic on the previous input and the new input, information on which pins rose and which fell can be extracted.

3.2 Port Initialization

Ports 1 and 2 were used to handle the sensor outputs. Each port has eight available pins, each capable of reading an external signal once an eight-pin header was soldered onto the target board. Ports 1 and 2 were specifically used because interrupts can only be enabled on these two ports for the CC430F5137. Pins 1–7 on Port 1 and pins 1–6 on Port 2 were configured as inputs with no internal pull-up resistor. The pins read a 0 V as a "0" and 3.3 V as a "1". One potential problem with this design is that until the sensor outputs are connected with the GND pad, which is actually 3.3 V, the outputs are floating values. Crosstalk between pins or other factors could cause false positives. A simple fix, if this does create a problem, would be to change the GND pad to give 0 V and initialize the pull-up resistors in the microcontroller. In this setup, 3.3 V

would be read as a “0” and 0 V would be a “1”. Interrupts can only be triggered in one direction at a time, so currently, it is selected to trigger when a low to high voltage transition occurs. The interrupt should only be serviced at the first input change, which should be a pin going from the initial state (“0”) to the changed state (“1”).

3.3 Real-time Clock Initialization

The real-time clock was used so that the time of the day could be recorded when an acceleration event occurred. The module was initialized with arbitrary values, but future versions will incorporate user input to set the clock.

3.4 Flash Initialization

Ten segments of flash memory were reserved to store the data and timestamps. These segments were allocated by writing them into the linker file. According to the data sheet, each segment of flash is accessible, by default, in blocks of 512 bytes. The program was designed so that each sample would consist of four bytes of data and two bytes for the time elapsed. Therefore, each flash segment could hold 85 samples, with 2 extra bytes. The last two bytes of each segment would be used as a status indicator. 0xEEEE marked a full segment, 0xFEFE marked a partially filled segment, and any other two bytes indicated that a blank or corrupted segment that needed to be erased. During initialization, the program would parse through the bank of segments, and search for the first empty or partially full segment, and point the position marker to the next available blank position.

3.5 Clock Initialization

In order to execute the data processing commands quickly enough, the main CPU clock was increased. TI code examples were used to change the CPU clock to either 1 or 12 MHz. The former was to be used during flash operations to reduce current draw tenfold, and the latter during processing operations to decrease cycle time. An internal signal called REFO, which is designed to run at 32,768 Hz, was used as a reference clock for the digitally controlled oscillator in place of an external crystal oscillator.

3.6 Low Power Mode and Interrupts

Until the first input is received, the CPU was turned off in a low power mode with general interrupts enabled to save power. The CC430 offers five low power modes, with higher modes turning off more processes. Low power mode 3 was selected because it turns off the CPU and main clock, but maintains power to ACLK, which sources the real-time clock. The CPU is restarted in less than 5 μs when any signal is read on the input pins (*7*).

3.7 Process Algorithm

After an input was read, the port interrupts were disabled. A software loop was used to poll for more samples rather than relying on interrupts to receive inputs because an interrupt-based

algorithm would suffer from interrupt latencies, which can be around 6 µs per interrupt. A 16-bit timer was set up sourced by the 12-MHz SMCLK divided by 4, in continuous mode. This timer was used to count the number of clock cycles between samples, which was then converted to a time in microseconds. For each cycle of the loop, the input pins were sampled, and if a change was detected, the four bytes of data and two bytes corresponding to the aforementioned timer were stored into an array in RAM. The timer was also reset, so the timer values corresponded to the number of clock cycles between samples. In addition, if the timer overflowed 25 times, then about 500 ms passed with no change in inputs, ending the loop. The timer could overflow less than 25 times, but still record a sample, i.e., an input changed after 20 ms but before the 500 ms window ended. In this case, the proper time would be the number of overflow occurrences multiplied by 20,000 µs and added to the timer value. However, completing this operation is time consuming and would also not fit in the 16 bits available. Instead, a special marker (0xFF00) plus the number of overflow occurrences was stored. An example of the timing of the sampling and data storage into RAM is shown in figure 8. The voltage level of the pulses (high vs. low) has no meaning in this plot; it is only the timing of the transitions that is important.

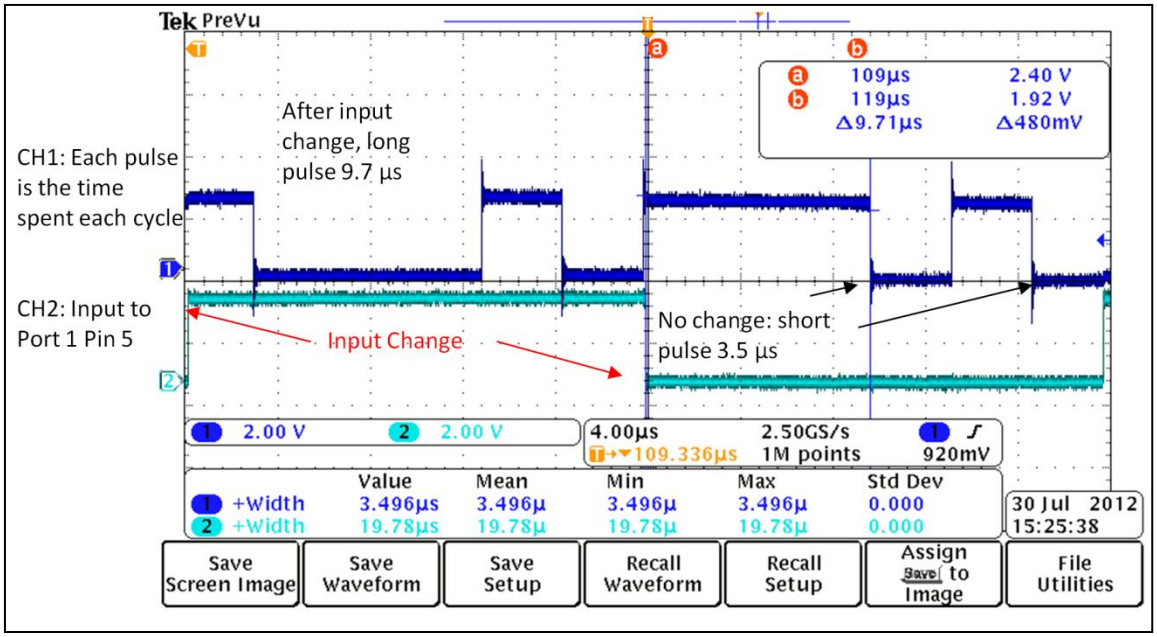

Figure 8. An oscilloscope image showing the timing of the sampling and data storage into RAM (CH1). The periodic input was used as input to one of the input pins (CH2).

3.8 Post-processing

Samples underwent post-processing before they were stored into flash and transmitted. Each sample consisted of six bytes, four of which corresponded to the data (i.e., which pins rose and which ones fell) and two to the timestamp. The timer value was converted to microseconds. The timer started at the end of one sampling cycle and ended when an input changed. It was found that 10 clock cycles passed during a sampling cycle with no input change, which corresponded to 3.5 µs, as seen in figure 8. The timer did not include the period of time needed to process a

sample, which was found to be 9.7 µs, as seen in figure 8. Therefore, the number of microseconds that elapsed was determined by using the timer value in the equation shown in equation 1.

$$\mathrm{T} = \mathrm{round}(\frac{3.5}{10} * timer + 9.7) \quad (1)$$

When applied to the sample in figure 8, where each input pulse was programmed to last 19.78 µs, the program evaluated the time to be 20 µs. However, consistently but infrequently, the time was evaluated to be 16 µs. This error is shown in figure 8, marked by the red label. On the left side of the trace, the first input change occurs very early in the sampling cycle, but after the actual sampling is complete, so the 3.5-µs cycle completes without a change registering. The next cycle is longer, as the change is registered and stored in RAM, which takes 9.7 µs. There are no input changes in the third and fourth cycles. The next change is registered in the fifth cycle. The actual time elapsed between the two changes is 19.78 µs, but since the program's time is based on the number of cycles that pass, the apparent time is 9.7 + 3.5*2 rounded down to 16 µs. This mistiming error can occur by chance, and so the 3.5-µs minimum sampling cycle time is the limiting factor for the accuracy of the timestamp. This error will be reduced by changing the method of calculation. The number of clock cycles will be used along with the frequency of the clock to calculate the time elapsed, which is more effective than using experimentally determined values to create a formula.

After filling a segment of flash memory with data, the flag bytes were changed to 0xEEEE, and the next available segment was accessed or erased. In the case where all segments were full, no more data would be written to flash, instead of overwriting previously full banks. This prioritized the first samples collected. In the event of a broken sensor or faulty circuitry, repeated nonsense data should not overwrite previous data. Afterwards, all variables in RAM were cleared, and the CPU was put into low power mode again.

4. Summary and Conclusions

The sampling delay was improved from the previous design. Previously, 100 µs were required between samples, causing data loss. With the new program, the delay has been improved to roughly 10 µs. The program has timestamp accuracy for each sample of about 3.5 µs. This means that, assuming samples are not input faster than the 10-µs delay, the program's evaluation of the time elapsed is accurate to within 3.5 µs.

There are still improvements that can be made to this program. It is incomplete, because RF transmission has not been implemented. The program will transmit samples that have just occurred in the last input cycle. It will also be able to send all data stored in flash upon user input. The timing delay will also be reduced after experimentation with different clock rates. The

use of an external crystal, which oscillates at more stable frequencies, especially in the case of varying conditions like high temperatures, will also be tested. Furthermore, the power consumption of the device will be optimized. For example, while decreasing the clock rate to 1 MHz for flash operation decreases the current tenfold, the time it takes to complete the process might also be 10 times faster. The real-time clock module will also be changed to implement user-input to set the time. Once the program is complete, device testing with the new generation sensors will be conducted.

5. References

1. Duma, S. M.; Manoogian, S. J.; Bussone, W. R.; Brolinson, P. G.; Goforth, M. W.; Donnenwerth, J. J.; Greenwald, R. M.; Chu, J. J.; Crisco, J. J. Analysis of Real-time Head Accelerations in Collegiate Football Players. *Clin J Sport Med.* **2005**, *15*, 3–8.
2. Sauser, B. Fighting Head Trauma in Iraq. *Technology Review*. MIT, 18 Sept. 2007. http://www.technologyreview.com/news/408688/fighting-head-trauma-in-iraq/ (accessed 16 Aug. 2012).
3. Smith, G. L.; Fan, L.; Balestrieri, R. E.; Jean, D. L. Micromechanical Shock Sensor, U.S. Patent US 6,737,979, May 18, 2004.
4. Currano, L. J.; Becker, C. R.; Lunking, D.; Smith, G. L.; Isaacson, B.; Thomas, L. Triaxial Inertial Switch with Multiple Thresholds and Resistive Ladder Readout. *Sensors and Actuators A: Physical*, in press. Retrieved from http://www.sciencedirect.com
5. Thomas, L. Personal communication, July 2012.
6. Davies, J. H. *MSP430 Microcontroller Basics*. Oxford: Newnes, 2008.
7. Texas Instruments, CC430 Family User's Guide, SLAU259B datasheet, May 2009 [Revised July 2010].

Appendix A. Pseudocode for the demo

```
#include "RF_Example.h"

#define  PACKET_LEN         (0x01)      // PACKET_LEN <= 61
#define  RSSI_IDX           (PACKET_LEN+1)  // Index of appended RSSI
#define  CRC_LQI_IDX        (PACKET_LEN+2)  // Index of appended LQI, checksum
#define  CRC_OK             (BIT7)          // CRC_OK bit
#define  PATABLE_VAL        (0x51)          // 0 dBm output

extern RF_SETTINGS rfSettings;

unsigned char packetReceived;
unsigned char packetTransmit;

unsigned char RxBuffer[64];
unsigned char RxBufferLength = 0;
const unsigned char TxBuffer[33]=
{PACKET_LEN, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06,
 0x07, 0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E,
 0x0F, 0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16,
 0x17, 0x18, 0x19, 0x1A, 0x1B, 0x1C, 0x1D, 0x1E,
 0x1F, 0x20};
const unsigned char TxBuffer2[2]= {PACKET_LEN, 0x02};
unsigned char buttonPressed = 0;
unsigned int i = 0;
unsigned char TxIndex = 0;

unsigned char transmitting = 0;
unsigned char receiving = 0;

char *base_addr = (char *)0x1880;
short * Flash_ptr = (short *) 0;                    // segC or segD
char * curPtr = (char *) 0;
char ptrOffset = 0;
unsigned char ptrDir = 1;                           // 1 = forward, 0 = backward

void erase_Flash (void);
void write_Seg (unsigned char[], unsigned char);
void setLED(char);


void main( void )
{
  // Stop watchdog timer to prevent time out reset
  WDTCTL = WDTPW + WDTHOLD;

  // Increase PMMCOREV level to 2 for proper radio operation
  SetVCore(2);

  ResetRadioCore();
  InitRadio();
```

```
  InitButtonLeds();

  ReceiveOn();
  receiving = 1;

  while (1)
  {
    __bis_SR_register( LPM3_bits + GIE );
    __no_operation();

    if (buttonPressed)                      // Process a button press->transmit
    {
      P3OUT |= BIT6;                        // Pulse LED during Transmit
      buttonPressed = 0;
      ReceiveOff();
      receiving = 0;
      unsigned char TxStuff[2];
      TxStuff[0] = PACKET_LEN;
      TxStuff[1] = TxBuffer[(TxIndex==0||TxIndex>31)?32:TxIndex];
      Transmit( (unsigned char*) TxStuff, sizeof TxStuff);

      transmitting = 1;

      P1IE |= BIT6;                         // Re-enable button press
    }
    else if(!transmitting)
    {
      ReceiveOn();
      receiving = 1;
    }
  }
}

void InitButtonLeds(void)
{
  // Set up the button as interruptible
  P1DIR = ~(BIT1+BIT2+BIT3+BIT4+BIT5+BIT6);
  P1REN |= (BIT1+BIT2+BIT3+BIT4+BIT5+BIT6);
  P1OUT |= (BIT1+BIT2+BIT3+BIT4+BIT5+BIT6);
  P1IE |= BIT6;
  P1IES |= BIT6;
  P1IFG &= ~BIT6;

  //P1.7 is GND
  //P1DIR.7 is already 1
  P1OUT &= ~BIT7;

  // Set up Port 2
  P2DIR = 0x3F;
  P2REN |= (BIT6+BIT7);
  P2OUT |= (BIT6+BIT7);
  P2OUT &= ~(BIT1+BIT2+BIT3+BIT4+BIT5);
  P2IE |= (BIT6+BIT7);
  P2IES |= (BIT6+BIT7);
  P2IFG &= ~(BIT6+BIT7);

  // Initialize Port J
```

```
  PJOUT = 0x00;
  PJDIR = 0xFF;
  // Set up LEDs
  P1OUT &= ~BIT0;
  P1DIR |= BIT0;
  P3OUT &= ~BIT6;
  P3DIR |= BIT6;
}

void InitRadio(void)
{
  // Set the High-Power Mode Request Enable bit so LPM3 can be entered
  // with active radio enabled
  PMMCTL0_H = 0xA5;
  PMMCTL0_L |= PMMHPMRE_L;
  PMMCTL0_H = 0x00;

  WriteRfSettings(&rfSettings);

  WriteSinglePATable(PATABLE_VAL);
}

void Transmit(unsigned char *buffer, unsigned char length)
{
  RF1AIES |= BIT9;
  RF1AIFG &= ~BIT9;                         // Clear pending interrupts
  RF1AIE |= BIT9;                           // Enable TX end-of-packet interrupt

  WriteBurstReg(RF_TXFIFOWR, buffer, length);

  Strobe( RF_STX );                         // Strobe STX
}

void ReceiveOn(void)
{
  RF1AIES |= BIT9;                          // Falling edge of RFIFG9
  RF1AIFG &= ~BIT9;                         // Clear a pending interrupt
  RF1AIE  |= BIT9;                          // Enable the interrupt

  // Radio is in IDLE following a TX, so strobe SRX to enter Receive Mode
  Strobe( RF_SRX );
}

void ReceiveOff(void)
{
  RF1AIE &= ~BIT9;                          // Disable RX interrupts
  RF1AIFG &= ~BIT9;                         // Clear pending IFG

  // It is possible that ReceiveOff is called while radio is receiving a packet.
  // Therefore, it is necessary to flush the RX FIFO after issuing IDLE strobe
  // such that the RXFIFO is empty prior to receiving a packet.
  Strobe( RF_SIDLE );
  Strobe( RF_SFRX  );
}

#pragma vector=CC1101_VECTOR
__interrupt void CC1101_ISR(void)
```

```
{
  switch(__even_in_range(RF1AIV,32))        // Prioritizing Radio Core Interrupt
  {
    case  0: break;                         // No RF core interrupt pending
    case  2: break;                         // RFIFG0
    case  4: break;                         // RFIFG1
    case  6: break;                         // RFIFG2
    case  8: break;                         // RFIFG3
    case 10: break;                         // RFIFG4
    case 12: break;                         // RFIFG5
    case 14: break;                         // RFIFG6
    case 16: break;                         // RFIFG7
    case 18: break;                         // RFIFG8
    case 20:                                // RFIFG9
      if(receiving)                     // RX end of packet
      {
        // Read the length byte from the FIFO
        RxBufferLength = ReadSingleReg( RXBYTES );
        ReadBurstReg(RF_RXFIFORD, RxBuffer, RxBufferLength);

        // Stop here to see contents of RxBuffer
        __no_operation();

        // Check the CRC results
        if(RxBuffer[CRC_LQI_IDX] & CRC_OK) {
            //if(RxBuffer[1]==0x20)
            //{
                    //#32 received
                    TA1CCTL0 = CCIE;
                    // CCR0 interrupt enabled
                    TA1CCR0 = 25000;
                    TA1CTL = TASSEL_2 + MC_2 + TACLR + ID_1;
                    // SMCLK, contmode, clear TAR
            //}
            write_Seg(RxBuffer, RxBufferLength-2);
        }

      }
      else if(transmitting)             // TX end of packet
      {
        RF1AIE &= ~BIT9;                // Disable TX end-of-packet interrupt
        P3OUT &= ~BIT6;                 // Turn off LED after Transmit
        transmitting = 0;
      }
      else while(1);                    // trap
      break;
    case 22: break;                         // RFIFG10
    case 24: break;                         // RFIFG11
    case 26: break;                         // RFIFG12
    case 28: break;                         // RFIFG13
    case 30: break;                         // RFIFG14
    case 32: break;                         // RFIFG15
  }
  __bic_SR_register_on_exit(LPM3_bits);
}

// Timer A0 interrupt service routine
```

```
#pragma vector=TIMER1_A0_VECTOR
__interrupt void TIMER1_A0_ISR(void)
{
  P1OUT ^= BIT0;                            // Toggle P1.0
  TA1CCR0 += 25000;                         // Add Offset to CCR0
  if(TA1CCR0 >= 60000)
        TA1CTL = MC_0;
}

void write_Seg (unsigned char buffer[], unsigned char length) {
      unsigned int i;
      __disable_interrupt();
                              // 5xx Workaround: Disable global
                              // interrupt while erasing. Re-Enable
                              // GIE if needed

      FCTL3 = FWKEY;                              // Clear Lock bit
      if(Flash_ptr == 0 || Flash_ptr == (short *)base_addr || Flash_ptr >=
(short *)(base_addr+2*(1+*(base_addr)))){
            erase_Flash();
            curPtr = base_addr;
            ptrOffset = 0;
      }
      setLED(buffer[1]);

      FCTL1 = FWKEY+WRT;
                                          // Set WRT bit for write operation

      for (i = 1; i < length; i++)        // length-2; do not want CRC_OK extra
      {
        *Flash_ptr++ = buffer[i];         // Write value to flash
      }
      FCTL1 = FWKEY;                            // Clear WRT bit
      FCTL3 = FWKEY+LOCK;                       // Set LOCK bit
      __enable_interrupt();                     // Enable global interrupt
      return;
}

void erase_Flash(){
      FCTL1 = FWKEY+ERASE;                    // Set Erase bit
      *Flash_ptr = 0;                         // Dummy write to erase Flash seg
      while(BUSY & FCTL3);
      base_addr = (char *)0x1880;
      Flash_ptr = (short *) base_addr;
      *Flash_ptr = 0;
      while(BUSY & FCTL3);
}


void setLED(char val){
  if(val >= 32) val = 0;
  P2OUT &= 0xC1;
  P2OUT |= (val << 1);
  __no_operation();
}

#pragma vector=PORT1_VECTOR
```

```
__interrupt void PORT1_ISR(void)
{
  switch(__even_in_range(P1IV, 16))
  {
    case  0: break;
    case  2: break;                           // P1.0 IFG
    case  4: break;                           // P1.1 IFG
    case  6: break;                           // P1.2 IFG
    case  8: break;                           // P1.3 IFG
    case 10: break;                           // P1.4 IFG
    case 12: break;                           // P1.5 IFG
    case 14:                                  // P1.6 IFG
        P1IE = 0;                             // Debounce by disabling buttons
        P1IFG &= ~BIT6;
        //P1IES ^= BIT6;
        buttonPressed = 1;
        TxIndex = (~(P1IN >> 1)) & ~(0xE0);
        __bic_SR_register_on_exit(LPM3_bits); // Exit active
      break;
    case 16: break;                           // P1.7 IFG
  }
}

#pragma vector=PORT2_VECTOR
__interrupt void PORT2_ISR(void)
{
  switch(__even_in_range(P2IV, 16))
  {
    case  0: break;
    case  2: break;                           // P2.0 IFG
    case  4: break;                           // P2.1 IFG
    case  6: break;                           // P2.2 IFG
    case  8: break;                           // P2.3 IFG
    case 10: break;                           // P2.4 IFG
    case 12: break;                           // P2.5 IFG
    case 14:                                  // P2.6 IFG
      P2IE &= ~BIT6;
      P2IES ^= BIT6;
      ptrDir ^= 1;
      if(ptrDir){
            P1OUT |= BIT0;
            P3OUT &= ~BIT6;
      } else {
            P1OUT &= ~BIT0;
            P3OUT |= BIT6;
      }
      P2IE |= BIT6;
      P2IFG &= ~BIT6;
        __bic_SR_register_on_exit(LPM3_bits); // Exit active
      break;
    case 16:                                  // P2.7 IFG
      P2IE &= ~BIT7;
      if(ptrDir)                            // go forward
      {
            if((short *)(base_addr+ptrOffset+2)<Flash_ptr)
                  ptrOffset+=2;
      } else {                              // go backward
```

```
            if(ptrOffset>=2)
                  ptrOffset -= 2;
      }
      setLED(*(base_addr+ptrOffset));
      P2IE |= BIT7;
      P2IFG &= ~BIT7;
        __bic_SR_register_on_exit(LPM3_bits); // Exit active
      break;
  }
}
```

INTENTIONALLY LEFT BLANK.

Appendix B. Code for Version 4

This is the latest version to-date (version 4).

```
/*
 * Timothy Lee
 * 8/1/2012
 *
 * Version 4 fully implements the flash storage of the sample and time stamp
 */

#include "cc430x513x.h"

//  Functions
void init_CLK(int);
void init_RTC(void);
void init_LED(void);
int init_Flash_sec(void);
void process(void);
void write_Seg();
void erase_Seg(unsigned short*);

//  Variables
unsigned short samp0[255] = { 0 };
unsigned short samp1[255] = { 0 };
unsigned short samp2[255] = { 0 };
unsigned short samp3[255] = { 0 };
unsigned short samp4[255] = { 0 };
unsigned short* sampBank[5] = { samp0, samp1, samp2, samp3, samp4 };
unsigned char sBankPtr = 0, sampPtr = 0;
char flag = 0;

//RF variables

//  Flash variables
char spaceAvail = 1;
#pragma DATA_SECTION(T0, ".mydata0");
#pragma DATA_ALIGN (T0, 2);  // reserve 2 bytes for each variable
unsigned short T0[256];
#pragma DATA_SECTION(T1, ".mydata1");
#pragma DATA_ALIGN (T1, 2);  // reserve 2 bytes for each variable
unsigned short T1[256];
#pragma DATA_SECTION(T2, ".mydata2");
#pragma DATA_ALIGN (T2, 2);  // reserve 2 bytes for each variable
unsigned short T2[256];
#pragma DATA_SECTION(T3, ".mydata3");
#pragma DATA_ALIGN (T3, 2);  // reserve 2 bytes for each variable
unsigned short T3[256];
#pragma DATA_SECTION(T4, ".mydata4");
#pragma DATA_ALIGN (T4, 2);  // reserve 2 bytes for each variable
unsigned short T4[256];
#pragma DATA_SECTION(T5, ".mydata5");
#pragma DATA_ALIGN (T5, 2);  // reserve 2 bytes for each variable
```

```
unsigned short T5[256];
#pragma DATA_SECTION(T6, ".mydata6");
#pragma DATA_ALIGN (T6, 2);  // reserve 2 bytes for each variable
unsigned short T6[256];
#pragma DATA_SECTION(T7, ".mydata7");
#pragma DATA_ALIGN (T7, 2);  // reserve 2 bytes for each variable
unsigned short T7[256];
#pragma DATA_SECTION(T8, ".mydata8");
#pragma DATA_ALIGN (T8, 2);  // reserve 2 bytes for each variable
unsigned short T8[256];
#pragma DATA_SECTION(T9, ".mydata9");
#pragma DATA_ALIGN (T9, 2);  // reserve 2 bytes for each variable
unsigned short T9[256];
unsigned short* bank[10] = { T0, T1, T2, T3, T4, T5, T6, T7, T8, T9 };
unsigned char curBankPtr = 0, curDataPtr = 0;
#define FULL 0xEEEE
#define INUSE 0xFEFE
#define DEFAULT 0xFFFF

void main(void) {
      WDTCTL = WDTPW + WDTHOLD; // Stop Watchdog Timer
      int i1=0, j1=0;
      for ( i1 = 0; i1 < 5; i1++)
            for( j1 = 0; j1<255; j1++)
                  sampBank[i1][j1] = 0;
      init_LED();
      init_RTC();
      spaceAvail = init_Flash_sec();
      init_CLK(12);
      // USE spaceAvail FLAG!

      //  Timer0 A0 will time how long the processing mode takes
      while (1) {

            //  A button input will wake the program up
            __bis_SR_register(LPM3_bits + GIE);
            // Enter LPM3, enable interrupts
            __no_operation(); // For debugger
            if ((P1IE || P2IE) == 0) {
                  //  The real time clock will reserve a space and add its
                  //  time value when process is done
                  //  RTC CODE HERE
                  TA0CTL = TASSEL_2 | MC_2 | ID_2 | TAIE | TACLR;
                  //  continuous mode, /4, interrupt enabled (overflow),
                  //  clear
                  process();
                  //  Clear timers
                  TA0CTL &= MC_0; //  Stop the processing timer

                  //  Program resulting data into flash
                  write_Seg();

                  //  Transmit data over RF

                  //  Clear variables
                  flag = 0;
                  int i=0, j=0;
```

```
                  for ( i = 0; i < sBankPtr+1; i++)
                        for( j = 0; j<256; j++)
                              sampBank[i][j] = 0;
                  sBankPtr = 0;
                  sampPtr = 0;
            }

            P1IE |= ~BIT0;
            P2IE |= (BIT1 + BIT2 + BIT3 + BIT4 + BIT5 + BIT6);
      } //end while
} // end main

//  Process method
//  Once awake, the software will poll the inputs for a certain length of time
//  ~500 ms
void process() {
      //  Samples are of the form
      //  00YYYYYYXXXXXXXXX0
      //  Ys are bits 1-6 from port 2. Xs are bits 1-7 from port 1
      volatile unsigned short lastIn = 0, newIn, temp, lastTime = 0;
      while (flag < 25) {
            //  P1 bits directly put into first 8 bits
            //  P2 bits shifted right once to eliminate pin 0, then left
            //  shifted 8 times to proper place
            //  Final is anded with 0011 1111 1111 1110 = 3FFE
            newIn = (P1IN + ((P2IN >> 1) << 8)) & 0x3FFE;
            //  XOR inputs to show differences
            if ((newIn ^ lastIn) != 0) {
                  if (sampPtr == 255) {
                        sampPtr = 0;
                        sBankPtr++;
                  }
                  //  If there are differences, then store the positive edge
                  //  results first, then negative edge results
                  //  newIn & ~lastIn gives positive edge results (+1
                  //  indicates pos edge)
                  //  lastIn & ~newIn gives negative edge results (0 on lsb
                  //  indicates neg edge)
                  sampBank[sBankPtr][sampPtr++] = (newIn & ~lastIn) + 1;
                  sampBank[sBankPtr][sampPtr++] = lastIn & ~newIn;
                  //  if samplePtr reaches X (500), then all spaces in the
                  //  array are full
                  //  Is this to be expected? samplePtr = 500 means 250
                  //  changes have been detected in a 500 ms time period
                  //  With 13 inputs, 250 allows for each input to change 20
                  //  times
                  TA0CTL &= MC_0;
                  //  Store the number of clock cycles since the last data
                  //  capture event
                  //  This will be 0 if this is a new capture cycle
                  //  If flag is not 0, then an overflow has occurred.  This
                  //  means the input is coming after a wait period of > 20
                  //  ms
                  //  A space will be filled with FF00 + flag so that it can
                  //  be checked during postprocessing
                  sampBank[sBankPtr][sampPtr++] = (flag) ? 0xFF00 + flag : TA0R;
                  //  Equation to convert each time from the timer value to
```

```
                  //  actual time periods is
                  //  time[timePtr] = (short) (3.4/10.5*(TA0R - 22));
                  //  However as this takes 20 us for each conversion, it
                  //  will be done in post-processing before writing to
                  //  flash/transmission
                  //  In addition, if data is continuously sent at a rate of
                  //  < 12 us, the timer will be periodically off.  In
                  //  addition, the time
                  //  obtained above is an estimate and can be off by a few
                  //  microseconds (error gets worse with more time).

                  //  Start processing timer
                  flag = 0;
                  TA0CTL = TASSEL_2 | MC_2 | ID_2 | TAIE | TACLR;
                  //  continuous mode, /4, interrupt enabled (overflow),
                  //  clear
            }
            lastIn = newIn;
            P3OUT ^= BIT1;
      } //  end while
      __no_operation(); //  For Debugging
} //  end Process

void write_Seg() {
      __disable_interrupt(); // 5xx Workaround: Disable global
                                    // interrupt while erasing. Re-Enable
                                    // GIE if needed
      // SET DCOCLK TO 1 MHZ AGAIN AND THEN RESET AFTER FLASH
      init_CLK(1);

      FCTL3 = FWKEY;
      FCTL1 = FWKEY + WRT;
      unsigned char i = 0, j = 0;
      //  Iterate through the samples
      for (i = 0; i < sBankPtr + 1; i++) {
            for (j = 0; j < 255; j += 3)
                  //  If there has been a measurement, the first of the 3
                  //  numbers
                  //  will end in a 1 (pos edge).  If it is 0, then this
                  //  space has not
                  //  been used, and we are done
                  if (sampBank[i][j] == 0)
                        j = 252;
                  else {
                        bank[curBankPtr][curDataPtr++] = sampBank[i][j];
                        while (FCTL3 & BUSY)
                              ;
                        bank[curBankPtr][curDataPtr++] = sampBank[i][j + 1];
                        while (FCTL3 & BUSY)
                              ;
                        bank[curBankPtr][curDataPtr++] = sampBank[i][j + 2];
                        while (FCTL3 & BUSY)
                              ;
                        //  If the segment is full, mark the segment and
                        //  increment the bank pointer
                        //  Initialize the new bank
                        if (curDataPtr == 255){
```

```
                              bank[curBankPtr++][curDataPtr++] = FULL;
                              init_Flash_sec();
                        }
                  }
      }
      FCTL1 = FWKEY;
      FCTL3 = FWKEY + LOCK;
      __enable_interrupt();

      //  Reset the clock to 12 MHz
      init_CLK(12);
} //  End write_Seg

//  erase_Seg erases a segment in Flash
//  It also writes 0xFEFE into the 255th location to indicate the segment is
//  IN USE
void erase_Seg(unsigned short *S) {
      __disable_interrupt();
      FCTL3 = FWKEY;
      FCTL1 = FWKEY + ERASE;
      *S = 0;
      while (FCTL3 & BUSY)
            ;
      FCTL1 = FWKEY + WRT;
      S[255] = 0xFEFE;
      FCTL1 = FWKEY; // Clear WRT bit
      FCTL3 = FWKEY + LOCK; // Set LOCK bit
} // end erase_Seg

//  Timer0_A5 CC1-4
//  This vector is triggered with TA0 TAIFG.
//  Set up to occur only with overflow events
#pragma vector = TIMER0_A1_VECTOR
__interrupt void TIMER0_A1_ISR(void) {
      TA0CTL = TASSEL_2 | MC_2 | ID_2 | TAIE | TACLR;
      //  continuous mode, /4, interrupt enabled (overflow), clear
      TA0CTL &= ~TAIFG;
      flag++;
} //  end Timer0 CC1-4 interrupt vector

//  Timer0_A5 CC0
//  This vector is triggered with TA0 CCR0 interrupt is triggered i.e. TA0CCR0
//  is reached
//  The interrupt is cleared automatically
#pragma vector = TIMER0_A0_VECTOR
__interrupt void TIMER0_A0_ISR(void) {
      __bic_SR_register_on_exit(LPM3_bits);
} //  end Timer0 CC0 interrupt vector

//  Port 1 Interrupt Vector
//  Can be used to wake up the processor in the event of a change in port 1
#pragma vector=PORT1_VECTOR
__interrupt void Port1_ISR(void) {
      P1IE = 0;
      P2IE = 0;
      while (P2IFG != 0 || P1IFG != 0) {
            P1IFG = 0;
```

```
            P2IFG = 0;
      }
      __bic_SR_register_on_exit(LPM3_bits);
} //  end port 1 interrupt vector

//  Port 2 Interrupt Vector
//  Can be used to wake up the processor in the event of a change in port 2
#pragma vector=PORT2_VECTOR
__interrupt void Port2_ISR(void) {
      P1IE = 0;
      P2IE = 0;
      while (P2IFG != 0 || P1IFG != 0) {
            P1IFG = 0;
            P2IFG = 0;
      }
      __bic_SR_register_on_exit(LPM3_bits);
} //  end port 2 interrupt vector

//  Real-Time Clock Interrupt Vector
#pragma vector=RTC_VECTOR
__interrupt void RTC_ISR(void) {
      switch (__even_in_range(RTCIV, 16)) {
      case 0:
            break; // No interrupts
      case 2: // RTCRDYIFG
            P3OUT ^= BIT6;
            RTCCTL01 &= ~RTCHOLD;
            RTCCTL01 &= ~RTCRDYIFG;
            break;
      case 4:
            break; // RTCEVIFG
      case 6:
            break; // RTCAIFG
      case 8: // RT0PSIFG
            RTCCTL01 &= ~RT0PSIFG;
            __bic_SR_register_on_exit(LPM3_bits);
            break;
      case 10: // RT1PSIFG
            RTCCTL01 &= ~RT1PSIFG;
            __bic_SR_register_on_exit(LPM3_bits);
            break;
      case 12:
            break; // Reserved
      case 14:
            break; // Reserved
      case 16:
            break; // Reserved
      default:
            break;
      } //  end case
} //  end Real-Time Clock Interrupt Vector

//  Real-Time Clock Initialization
void init_RTC() {
      RTCCTL01 = RTCHOLD | RTCMODE; //  hex-mode, hold, calendar mode
//  Base clock = ACLK = 32kHz
//  RT0 clock /256
```

```
//  RT1 clock RT0 /128
      RTCPS1CTL |= RT1PSIE + RT1IP_6;
//  Enable RT1 Interrupt, Prescaler set /128
      RTCTIM0 = 0x0000; //  0 sec 0 min
      RTCHOUR = 0x11; //  17 hour
      RTCDOW = 0; //  Day of the week - 0
      /*
      RTCDATE = 0x0719; //  Date 7/19
      RTCYEAR = 0x2012; //  Year 2012
      */
      RTCCTL01 &= ~RTCHOLD; //  Activate RTC
} // end RTC

//  Clock Initialization
void init_CLK(int n) {
      UCSCTL3 |= SELREF_2; // Set DCO FLL reference = REFO
      UCSCTL4 |= SELA_2; // Set ACLK = REFO

      __bis_SR_register(SCG0);
// Disable the FLL control loop
      UCSCTL0 = 0x0000; // Set lowest possible DCOx, MODx
      if (n == 12) {
            UCSCTL1 = DCORSEL_5; // Select DCO range 24MHz operation
            UCSCTL2 = FLLD_1 + 374; // Set DCO Multiplier for 12MHz
      } else if (n == 1) {
            UCSCTL1 = DCORSEL_1;
            UCSCTL2 = FLLD_1 + 30;
      }
// (N + 1) * FLLRef = Fdco
// (374 + 1) * 32768 = 12MHz
// Set FLL Div = fDCOCLK/2
      __bic_SR_register(SCG0);
// Enable the FLL control loop

// Worst-case settling time for the DCO when the DCO range bits have been
// changed is n x 32 x 32 x f_MCLK / f_FLL_reference. See UCS chapter in 5xx
// UG for optimization.
// 32 x 32 x 12 MHz / 32,768 Hz = 375000 = MCLK cycles for DCO to settle
      if (n == 1)
            __delay_cycles(31250);
      else if (n == 12)
            __delay_cycles(375000);


// Loop until XT1,XT2 & DCO fault flag is cleared
      do {
            UCSCTL7 &= ~(XT2OFFG + XT1LFOFFG + XT1HFOFFG + DCOFFG);
            // Clear XT2,XT1,DCO fault flags
            SFRIFG1 &= ~OFIFG; // Clear fault flags
      } while (SFRIFG1 & OFIFG); // Test oscillator fault flag
}

//  LED Initialization
void init_LED() {
//  Port 1.0 Green LED, Port 3.6 Red LED
//  Port 1.1-1.7 inputs for levels 1-7
//  Port 2.1-2.6 inputs for level 8, Z, Y, X, NEG, POS
```

```
//  Port 2.7 OUTPUT GND
//  0 is Input, 1 is Output
      P1DIR = BIT0;
      P2DIR = ~(BIT1 + BIT2 + BIT3 + BIT4 + BIT5 + BIT6);
      P3DIR |= BIT6;

      P2OUT |= (BIT1 + BIT2 + BIT3 + BIT4 + BIT5 + BIT6);
      P2OUT &= ~BIT7;
//  Enable Interrupts from low to high
      P1IES &= BIT0;
      P2IES &= ~(BIT1 + BIT2 + BIT3 + BIT4 + BIT5 + BIT6);
      P1IE |= ~BIT0;
      P2IE |= (BIT1 + BIT2 + BIT3 + BIT4 + BIT5 + BIT6);
//  Clear IFG
      P1IFG = 0;
      P2IFG = 0;
//  LEDs
      P3DIR |= BIT6;
      P3OUT &= ~BIT6;
      P1OUT &= ~BIT0; //  Green LED

//TESTING PURPOSES FOR THIRD VERSION
      P3DIR |= BIT1;
      P3OUT |= BIT1;
      P3OUT &= ~BIT1;

} //  end LED Initialization

//  init_Flash_sec goes through the bank to initialize the bank and data pointer
//  If the segment is marked as in use, then iterate through to retrieve the
//  next available data pointer
//  If the segment is marked as full, go to the next segment
//  If the segment is marked as blank, erase the bank
int init_Flash_sec() {
      unsigned char i = 0, j = 0;
      //  Iterate through at most 10 banks
      while (i < 10) {
            //  If the ith segment's last entry is 0xFFFF, then it is an
            //  empty segment
            //  Erase the segment so it can be written to, set the pointers
            //  to 0
            //  Return a 1 indicating space is available
            //  If the ith segment's last entry is 0xFEFE, then the segment
            //  has been erased
            //  It may be partially full, so iterate through until a default
            //  byte is found
            if (bank[i][255] == INUSE) {
                  curBankPtr = i;
                  for (j = 0; j < 255; j += 3)
                        if (bank[i][j] == DEFAULT) {
                              curDataPtr = j;
                              return 1;
                        }
                  //  If no data points are found, then the data is corrupt
                  //  Ignore this segment and increment to the next
                  i++;
            }
```

```
            //  If the ith segment's last entry is 0xEEEE, then the segment
            //  is full
            //  Increment the index to the next segment, as no more room is
            //  available in this seg
            else if (bank[i][255] == FULL)
                  i++;
            else
            //  If the last entry is none of the above, then the flash memory
            //  is corrupt or blank
            //  Treat as if it were a DEFAULT
            {
                  erase_Seg(bank[i]);
                  curBankPtr = i;
                  curDataPtr = 0;
                  return 1;
            }
      }
      //  If none of the segments are in use or reset, then there is no flash
      //  available
      return 0;
} //  end init_Flash_sec
```

1 (PDF only) DEFENSE TECHNICAL INFORMATION CTR
DTIC OCA
8725 JOHN J KINGMAN RD
STE 0944
FORT BELVOIR VA 22060-6218

1 DIRECTOR
US ARMY RESEARCH LAB
IMAL HRA
2800 POWDER MILL RD
ADELPHI MD 20783-1197

1 DIRECTOR
US ARMY RESEARCH LAB
RDRL CIO LL
2800 POWDER MILL RD
ADELPHI MD 20783-1197

1 DIRECTOR
US ARMY RESEARCH LAB
RDRL CIO LT
2800 POWDER MILL RD
ADELPHI MD 20783-1197

4 DR. LUKE CURRANO
U.S. ARMY RESEARCH LAB
RDRL SER L
2800 POWDER MILL RD
ADELPHI MD 20783-1197

1 JOHN PAUL KRUSZEWSKI
NSRDEC
14 KANSAS ST
NATICK, MA 01760

1 DON LEE
NSRDEC
14 KANSAS ST
NATICK, MA 01760

1 KARL MASTERS
PM SEQ
10170 BEACH ROAD
FORT BELVOIR, VA 22060

1 CHRIS PERRITT
PM SEQ
10170 BEACH ROAD
FORT BELVOIR, VA 22060